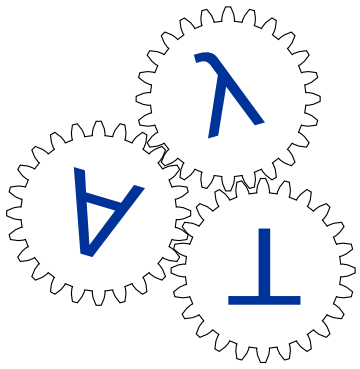
GAPT

General Architecture for Proof Theory



User manual

Version 1.11-SNAPSHOT

December 15, 2015

List of Corrections

Short example for first and higher-order CERES. 16
Expansion trees? 26

Contents

1 Introduction **3**

2 Download and execution **3**
2.1 System requirements 4

3 Data structures **4**
3.1 Formulas 4
3.2 Proofs 5

4 Feature walkthrough **6**
4.1 SAT solver interface 6
4.2 MaxSAT solver interface 8
4.3 SMT solver interface 9
4.4 First-order theorem prover interface 10
4.5 Built-in tableaux prover 12
4.6 Cut-elimination (Gentzen's method) 12
4.7 Skolemization 13
4.8 Interpolation 14
4.9 Expansion trees 15
4.10 Cut-elimination by resolution (CERES) 16
4.11 Cut-introduction 16
4.12 Tree grammars 18

A Proof systems **20**
A.1 LK 20
A.2 Resolution 22
A.3 LKsk 23
A.4 Ral 25

B Prover9 term parser (parseFormula) **26**

B.1 LADR naming scheme (default) 26

B.2 Prolog naming scheme 26

B.3 Terms and Formulas 27

C XML proof parser **27**

1 Introduction

GAPT is a generic architecture for proof transformations implemented in Scala.

The focus of GAPT are proof transformations (in contrast to proof assistants, whose focus is proof formalization, and automated deduction systems, whose focus is proof search). GAPT is used from a shell that provides access to the functionality in the system in a way that is inspired by computer algebra systems: the basic objects are formulas and (different kinds of) proofs which can be modified by calling GAPT commands from the command line. In addition, there is a graphical user interface that allows the user to view (and—to a certain extent— modify) proofs in a flexible and visually appealing way.

The current functionality of GAPT includes data structures for formulas, sequents, resolution proofs, sequent calculus proofs, expansion tree proofs and algorithms for e.g. unification, proof Skolemization, cut-elimination, cut elimination by resolution [2], cut-introduction [5], etc.

2 Download and execution

There are three ways you can obtain GAPT:

1. **The recommended way:** You can download a package of the current version of GAPT at https://logic.at/gapt/. After extracting the `tar.gz`-file, you will find a shell script `gapt.sh`.

   Running this script will start the command line interface of GAPT:

```
./gapt.sh
```

2. If you are adventurous, you can also download an unstable development version from github:

```
git clone https://github.com/gapt/gapt
cd gapt
sbt console
```

3. If you like GAPT and want to use it as a library in your Scala project, it is available as a Maven artifact on JCenter. All you need to do is add one line to your `build.sbt`:

```
libraryDependencies += "at.logic.gapt" %% "gapt" % "1.10"
```

The command line interface of GAPT is an interactive Scala shell. This means that all functionality of Scala is available to you. In particular it is easy to write Scala scripts that use the functionality of GAPT.

You don't need to know anything about Scala to try out the examples in this manual, but if you do want to learn more about Scala we recommend the book "Programming in Scala" [9].

Interactions with the Scala shell are typeset in the following way:

```
gapt> println("Hello, world!")
Hello, world!
```

Here, **println("Hello, world!")** is the user input, and `Hello, world!` is the output from the Scala shell.

If you want to consult the in-depth API documentation of a function, you can use the `help` command:

```
gapt> help(containsQuantifierOnLogicalLevel)
```

2.1 System requirements

To run GAPT you need to have Java 7 (or higher) installed.

GAPT contains interfaces to the following automated reasoning systems. Installing them is optional. If GAPT does not find the executables in the path, the functionality of these systems will not be available.

- Prover9 (http://www.cs.unm.edu/~mccune/mace4/download/) - make sure the commands `prover9` and `prooftrans` are available.
- E theorem prover (http://eprover.org/)
- LeanCoP (http://leancop.de/)
- VeriT (http://www.verit-solver.org/)
- Z3 (https://github.com/Z3Prover/z3)
- MiniSAT (http://minisat.se/)
- Glucose (http://www.labri.fr/perso/lsimon/glucose/)
- Sat4J (http://sat4j.org/)
- OpenWBO (http://sat.inesc-id.pt/open-wbo/)
- CVC4 (http://cvc4.cs.nyu.edu/web/)

3 Data structures

3.1 Formulas

Formulas, terms, and other expressions are are represented as lambda terms in simple type theory. Terms and formulas of first-order logic and schematic first-order logic are hence encoded as lambda terms, these form regular subsets.

You can enter formulas by parsing them with the prover9 [7] parser:

```
gapt> val H = parseFormula( "(all x (P(x,f(x)) -> (exists y P(x,y))))" )
H: at.logic.gapt.expr.FOLFormula = ∀x.(P(x,f(x))⊃∃y.P(x,y))
```

The prover9 syntax was also extended to higher-order logic, where type declarations are added:

```
gapt> val I = parseLLKFormula ( "var P:o>i>o; const f:o>i; var x:o; var y:i; (all x (
    P(x,f(x))) -> (exists y P(x,y)))" )
I: at.logic.gapt.expr.HOLFormula = ∀x:o.(P(x,f(x))⊃∃y.P(x,y))
```

Please refer to Appendix B for a full description of the languages the parsers accept.

A collection of formula sequences can be found in the file `examples/FormulaSequences.scala`. Have a look at this code to see how to compose formulas without the parser. You can generate instances of these formula sequences by entering, e.g.,

```
gapt> val f = BussTautology( 5 )
f: at.logic.gapt.proofs.HOLSequent = ((((((c_1∨d_1)∧(c_2∨d_2))∧(c_3∨d_3))∧(c_4∨d_4)
    )⊃c_5)∨(((((c_1∨d_1)∧(c_2∨d_2))∧(c_3∨d_3))∧(c_4∨d_4))⊃d_5)), (((((c_1∨d_1)∧(
    c_2∨d_2))∧(c_3∨d_3))⊃c_4)∨((((c_1∨d_1)∧(c_2∨d_2))∧(c_3∨d_3))⊃d_4)), ((((c_1∨
    d_1)∧(c_2∨d_2))⊃c_3)∨(((c_1∨d_1)∧(c_2∨d_2))⊃d_3)), (((c_1∨d_1)⊃c_2)∨((c_1∨d_1)
    ⊃d_2)), (c_1∨d_1) :- c_5, d_5
```

3.2 Proofs

There are various possibilities for entering proofs into the system. The most basic one is a direct top-down proof-construction using the constructors of the inference rules.

```
gapt> val A = FOLAtom("A")
A: at.logic.gapt.expr.FOLAtom = A

gapt> val B = FOLAtom("B")
B: at.logic.gapt.expr.FOLAtom = B

gapt> val F1 = B --> (A & B)
F1: at.logic.gapt.expr.FOLFormula = (B⊃(A∧B))

gapt> val F2 = A & B
F2: at.logic.gapt.expr.FOLFormula = (A∧B)
```

We start with the axioms:

```
gapt> val p1 = LogicalAxiom(A)
p1: at.logic.gapt.proofs.lkNew.LogicalAxiom =
[p1] A :- A   (LogicalAxiom(A))

gapt> val p2 = LogicalAxiom(B)
p2: at.logic.gapt.proofs.lkNew.LogicalAxiom =
[p1] B :- B   (LogicalAxiom(B))
```

These are joined by an $\wedge : \mathrm{right}$-inference. See Appendix A.1 for the formal definition of the sequent calculus used in GAPT.

```
gapt> val p3 = AndRightRule( p1, A, p2, B )
p3: at.logic.gapt.proofs.lkNew.AndRightRule =
[p3] A, B :- (A∧B)  (AndRightRule(p1, Suc(0), p2, Suc(0)))
[p2] B :- B   (LogicalAxiom(B))
[p1] A :- A   (LogicalAxiom(A))
```

To finish the proof it remains to apply two $\supset$: right-inferences:

```
gapt> val p4 = ImpRightRule( p3, B, F2 )
p4: at.logic.gapt.proofs.lkNew.ImpRightRule =
[p4] A :- (B⊃(A∧B)) (ImpRightRule(p3, Ant(1), Suc(0)))
[p3] A, B :- (A∧B)  (AndRightRule(p1, Suc(0), p2, Suc(0)))
[p2] B :- B   (LogicalAxiom(B))
[p1] A :- A   (LogicalAxiom(A))

gapt> val p5 = ImpRightRule( p4, A, F1 )
p5: at.logic.gapt.proofs.lkNew.ImpRightRule =
[p5]  :- (A⊃(B⊃(A∧B))) (ImpRightRule(p4, Ant(0), Suc(0)))
[p4] A :- (B⊃(A∧B)) (ImpRightRule(p3, Ant(1), Suc(0)))
[p3] A, B :- (A∧B)  (AndRightRule(p1, Suc(0), p2, Suc(0)))
[p2] B :- B   (LogicalAxiom(B))
[p1] A :- A   (LogicalAxiom(A))
```

You can now view this proof by typing:

```
gapt> prooftool( p5 )
```

The system comes with a collection of example proof sequences in the file `examples/ProofSequences.scala` which are generated in the above style. Have a look at this code for more complicated proof constructions.

4 Feature walkthrough

4.1 SAT solver interface

The following shows an example session, using the Sat4j SAT solver to verify validity and satisfiability, and query the thus obtained models. Consider the *pigeon hole principle for* (m, n), $\mathrm{PHP}_{m,n}$, which states that if m pigeons are put into n holes, then there is a hole which contains two pigeons. It is valid iff $m > n$. $\neg\mathrm{PHP}_{m,n}$ states that when putting m pigeons into n holes, there is no hole containing two pigeons. This is satisfiable iff $m \leq n$.[1]

```
gapt> Sat4j isValid PigeonHolePrinciple(3, 2)
res3: Boolean = true
```

shows that $\mathrm{PHP}_{3,2}$ is valid, and

```
gapt> Sat4j isValid PigeonHolePrinciple(3, 3)
res4: Boolean = false
```

[1] In Scala, `Sat4j isValid formula` is syntactic sugar for `Sat4j.isValid(formula)`.

shows that $\mathrm{PHP}_{3,3}$ is not valid. Furthermore,

```
gapt> val Some(m) = Sat4j solve -PigeonHolePrinciple(3, 3)
m: at.logic.gapt.models.Interpretation =
R(p_1,h_1) -> false
R(p_1,h_2) -> false
R(p_1,h_3) -> true
R(p_2,h_1) -> true
R(p_2,h_2) -> false
R(p_2,h_3) -> false
R(p_3,h_1) -> false
R(p_3,h_2) -> true
R(p_3,h_3) -> false
```

yields a model of $\neg\mathrm{PHP}_{3,3}$ that can be queried:

```
gapt> val p1 = PigeonHolePrinciple.atom(1, 1)
p1: at.logic.gapt.expr.FOLAtom = R(p_1,h_1)

gapt> val p2 = PigeonHolePrinciple.atom(2, 1)
p2: at.logic.gapt.expr.FOLAtom = R(p_2,h_1)

gapt> m.interpret(p1) // Is pigeon 1 in hole 1?
res5: Boolean = false

gapt> m.interpret(p2) // Is pigeon 2 in hole 1?
res6: Boolean = true
```

We can also interpret quantifier-free formulas:

```
gapt> m.interpret( And(p1, p2) )
res7: Boolean = false
```

We can also convert $\neg\mathrm{PHP}_{3,3}$ into DIMACS format:

```
gapt> val (cnf, _, _) = structuralCNF(-PigeonHolePrinciple(3,3),
    generateJustifications=false, propositional=true)
cnf: Set[at.logic.gapt.proofs.HOLClause] = Set( :- R(p_3,h_1), R(p_3,h_2), R(p_3,h_3)
    , R(p_3,h_2), R(p_2,h_2) :- , R(p_2,h_3), R(p_1,h_3) :- , R(p_3,h_3), R(p_2,h_3)
    :- , :- R(p_2,h_1), R(p_2,h_2), R(p_2,h_3), R(p_3,h_3), R(p_1,h_3) :- , R(p_2,h_1
    ), R(p_1,h_1) :- , R(p_3,h_2), R(p_1,h_2) :- , R(p_3,h_1), R(p_1,h_1) :- , R(p_3,
    h_1), R(p_2,h_1) :- , R(p_2,h_2), R(p_1,h_2) :- , :- R(p_1,h_1), R(p_1,h_2), R(
    p_1,h_3))

gapt> val encoding = new DIMACSEncoding
encoding: at.logic.gapt.formats.dimacs.DIMACSEncoding = DIMACSEncoding()

gapt> writeDIMACS(encoding encodeCNF cnf)
res8: String =
"p cnf 9 12
1 2 3 0
-2 -4 0
```

```
-5 -6 0
-3 -5 0
7 4 5 0
-3 -6 0
-7 -8 0
-2 -9 0
-1 -8 0
-1 -7 0
-4 -9 0
8 9 6 0
"
```

If you want to know which variable in the DIMACS output corresponds to which atom in GAPT, you can query the `DIMACSEncoding` object:

```
gapt> encoding decodeAtom 1
res9: at.logic.gapt.expr.HOLAtom = R(p_3,h_1)
```

GAPT also supports other SAT solvers such as MiniSAT or Glucose out of the box:

```
gapt> MiniSAT isValid PigeonHolePrinciple(3,2)
res10: Boolean = true
```

```
gapt> Glucose isValid PigeonHolePrinciple(3,2)
res11: Boolean = true
```

If you have another DIMACS-compliant solver installed or want to pass extra options to the SAT solver, you can pass a custom command to GAPT as well:

```
gapt> val solver = new ExternalSATSolver("minisat", "-mem-lim=1024")
solver: at.logic.gapt.provers.sat.ExternalSATSolver = ExternalSATSolver("minisat", "-
    mem-lim=1024")

gapt> solver isValid PigeonHolePrinciple(3,2)
res12: Boolean = true
```

4.2 MaxSAT solver interface

The MaxSAT interface supports generating optimal solutions for weighted partial MaxSAT instances: these consist of a list of hard clauses, which must be satisfied in the solution; and a list of weighted soft clauses, where weight of the satisfied soft clauses must be maximized. See [1] for an overview.

Let us solve a simple example using the MaxSAT solver from SAT4J:

```
gapt> val Seq(a,b,c) = Seq("a","b","c") map {FOLAtom(_)}
a: at.logic.gapt.expr.FOLAtom = a
b: at.logic.gapt.expr.FOLAtom = b
c: at.logic.gapt.expr.FOLAtom = c

gapt> MaxSat4j.solve(hard = a|b|c, soft = Seq(-a -> 4, -b -> 3))
```

```
res13: Option[at.logic.gapt.models.Interpretation] =
Some(a -> false
b -> false
c -> true)
```

GAPT also supports other MaxSAT solvers out of the box, just write `OpenWBO` or `ToySolver` instead of `MaxSat4j`.

4.3 SMT solver interface

The SMT solver interface in GAPT supports validity queries for `QF_UF` formulas. For example we can check whether a quantifier-free formula is a quasi-tautology using veriT:

```
gapt> val f = parseFormula("(a=b | a=c) & P(c) & P(b) -> P(a)")
f: at.logic.gapt.expr.FOLFormula = (((a=b∨a=c)∧(P(c)∧P(b)))⊃P(a))
```

```
gapt> VeriT isValid f
res14: Boolean = true
```

GAPT also supports Z3 and CVC4 out of the box (if they are installed):

```
gapt> Z3 isValid f
res15: Boolean = true

gapt> CVC4 isValid f
res16: Boolean = true
```

You can export `QF_UF` formulas (or sequents) as SMT-LIB benchmarks; note that we apply a drastic renaming to the constant symbols in order to support arbitrary (even Unicode) names in GAPT:

```
gapt> val (benchmark, typeRenaming, constantRenaming) =
    SmtLibExporter(Sequent() :+ f)
benchmark: String =
"(set-logic QF_UF)
(declare-sort t1 0)
(declare-fun f1 (t1) Bool)
(declare-fun f2 () t1)
(declare-fun f3 () t1)
(declare-fun f4 () t1)
(assert (not (=> (and (or (= f4 f2) (= f4 f3)) (and (f1 f3) (f1 f2))) (f1 f4))))
(check-sat)
"
typeRenaming: Map[at.logic.gapt.expr.TBase,at.logic.gapt.expr.TBase] = Map(o -> Bool,
     i -> t1)
constantRenaming: Map[at.logic.gapt.expr.Const,at.logic.gapt.expr.Const] = Map(a ->
    f4, P -> f1, b -> f2, c -> f3)
```

We can also extract instances for basic equality axioms (reflexivity, symmetry, and congruences) from veriT's proof output:

```
gapt> val Some(expansionSequent) = VeriT getExpansionSequent (Sequent() :+ f)
expansionSequent: at.logic.gapt.proofs.expansionTrees.ExpansionSequent =
    WeakQuantifier(∀x.∀y.(x=y⊃y=x), ArrayBuffer((WeakQuantifier(∀y.(a=y⊃y=a),
    ArrayBuffer((Atom(a=b)⊃Atom(b=a),b), (Atom(a=c)⊃Atom(c=a),c))),a))),
    WeakQuantifier(∀x1.∀y1.((x1=y1∧P(x1))⊃P(y1)), ArrayBuffer((WeakQuantifier(∀y1.((
    c=y1∧P(c))⊃P(y1)), ArrayBuffer((Atom(c=a)∧Atom(P(c))⊃Atom(P(a)),a))),c), (
    WeakQuantifier(∀y1.((b=y1∧P(b))⊃P(y1)), ArrayBuffer((Atom(b=a)∧Atom(P(b))⊃Atom(P
    (a)),a))),b))) :- Atom(a=b)∨Atom(a=c)∧Atom(P(c))∧Atom(P(b))⊃Atom(P(a))

gapt> extractInstances(expansionSequent) foreach println
(a=b⊃b=a)
(a=c⊃c=a)
((c=a∧P(c))⊃P(a))
((b=a∧P(b))⊃P(a))
(((a=b∨a=c)∧(P(c)∧P(b)))⊃P(a))
```

4.4 First-order theorem prover interface

GAPT includes interfaces to several first-order theorem provers, such as Prover9, E prover, and LeanCoP. For Prover9 and E prover we can read back resolution proofs, and construct LK and expansion proofs from them. The LeanCoP interface only supports expansion sequent extraction.

Here is how you can get all of these kinds of proofs using Prover9:

```
gapt> val sequent = existsclosure("p(0)" +: "p(x) -> p(s(x))" +: Sequent()
                          :+ "p(s(s(0)))" map parseFormula)
sequent: at.logic.gapt.proofs.HOLSequent = p(0), ∀x.(p(x)⊃p(s(x))) :- p(s(s(0)))
```

```
gapt> Prover9 isValid sequent
res18: Boolean = true

gapt> Prover9 getRobinsonProof sequent
res19: Option[at.logic.gapt.proofs.resolution.ResolutionProof] =
Some([p11] :-    (Resolution(p6, Suc(0), p10, Ant(0)))
[p10] p(s(0)) :-   (Resolution(p7, Suc(0), p9, Ant(0)))
[p9] p(s(s(0))) :-   (Instance(p8, Substitution()))
[p8] p(s(s(0))) :-   (InputClause(p(s(s(0))) :- ))
[p7] p(s(0)) :- p(s(s(0))) (Instance(p4, Substitution(v0 -> s(0))))
[p6]  :- p(s(0))   (Resolution(p2, Suc(0), p5, Ant(0)))
[p5] p(0) :- p(s(0))  (Instance(p4, Substitution(v0 -> 0)))
[p4] p(v0) :- p(s(v0))  (Instance(p3, Substitution(x -> v0)))
[p3] p(x) :- p(s(x))  (InputClause(p(x) :- p(s(x))))
[p2]  :- p(0)   (Instance(p1, Substitution()))
[p1]  :- p(0)   (InputClause( :- p(0)))
)

gapt> Prover9 getLKProof sequent
res20: Option[at.logic.gapt.proofs.lkNew.LKProof] =
```

```
Some([p19] ∀x.(p(x)⊃p(s(x))), p(0) :- p(s(s(0))) (ContractionLeftRule(p18, Ant(2),
    Ant(1)))
[p18] p(0), ∀x.(p(x)⊃p(s(x))), ∀x.(p(x)⊃p(s(x))) :- p(s(s(0))) (CutRule(p9, Suc(0),
    p17, Ant(1)))
[p17] ∀x.(p(x)⊃p(s(x))), p(s(0)) :- p(s(s(0))) (CutRule(p14, Suc(0), p16, Ant(0)))
[p16] p(s(s(0))) :- p(s(s(0))) (ContractionRightRule(p15, Suc(0), Suc(1)))
[p15] p(s(s(0))) :- p(s(s(0))), p(s(s(0))) (WeakeningRightRule(p10, p(s(s(0)))))
[p14] ∀x.(p(x)⊃p(s(x))), p(s(0)) :- p(s(s(0))) (ContractionLeftRule(p13, Ant(0), Ant
    (1)))
[p13] ∀x.(p(x)⊃p(s(x))), ∀x.(p(x)⊃p(s(x))), p(s(0)) :- p(s(s(0))) (WeakeningLeftRule
    (p12, ∀x.(p(x)⊃p(s(x)))))
[p12] ∀x.(p(x)⊃p(s(x))), p(s(0)) :- p(s(s(0))) (ForallLeftRule(p11, Ant(0), (p(x)⊃p(
    s(x))), ...
gapt> Prover9 getExpansionSequent sequent
res21: Option[at.logic.gapt.proofs.expansionTrees.ExpansionSequent] = Some(Atom(p(0))
    , WeakQuantifier(∀x.(p(x)⊃p(s(x))), List((Atom(p(0))⊃Atom(p(s(0))),0), (Atom(p(s
    (0)))⊃Atom(p(s(s(0)))),s(0)))) :- Atom(p(s(s(0)))))
```

All of the above works with E prover as well, we will just show `getLKProof` as an example:

```
gapt> EProver getLKProof sequent
res22: Option[at.logic.gapt.proofs.lkNew.LKProof] =
Some([p19] p(0), ∀x.(p(x)⊃p(s(x))) :- p(s(s(0))) (CutRule(p3, Suc(0), p18, Ant(1)))
[p18] ∀x.(p(x)⊃p(s(x))), p(0) :- p(s(s(0))) (ContractionLeftRule(p17, Ant(2), Ant(0))
    )
[p17] ∀x.(p(x)⊃p(s(x))), p(0), ∀x.(p(x)⊃p(s(x))) :- p(s(s(0))) (CutRule(p8, Suc(0),
    p16, Ant(1)))
[p16] ∀x.(p(x)⊃p(s(x))), p(s(0)) :- p(s(s(0))) (CutRule(p13, Suc(0), p15, Ant(0)))
[p15] p(s(s(0))) :- p(s(s(0))) (ContractionRightRule(p14, Suc(0), Suc(1)))
[p14] p(s(s(0))) :- p(s(s(0))), p(s(s(0))) (WeakeningRightRule(p9, p(s(s(0)))))
[p13] ∀x.(p(x)⊃p(s(x))), p(s(0)) :- p(s(s(0))) (ContractionLeftRule(p12, Ant(0), Ant
    (1)))
[p12] ∀x.(p(x)⊃p(s(x))), ∀x.(p(x)⊃p(s(x))), p(s(0)) :- p(s(s(0))) (WeakeningLeftRule
    (p11, ∀x.(p(x)⊃p(s(x)))))
[p11] ∀x.(p(x)...
```

Note that `getLKProof` only works for sequents without strong quantifiers (i.e. sequents that are already Skolemized); however `getExpansionSequent` will happily return expansion sequents with Skolem quantifiers in that case:

```
gapt> val strong = ("(exists x all y P(x,y))" +: Sequent()
    :+ "(all y exists x P(x,y))" map parseFormula)
strong: at.logic.gapt.proofs.Sequent[at.logic.gapt.expr.FOLFormula] = ∃x.∀y.P(x,y) :-
     ∀y.∃x.P(x,y)

gapt> Prover9 getExpansionSequent strong
res23: Option[at.logic.gapt.proofs.expansionTrees.ExpansionSequent] = Some(
    SkolemQuantifier(∃x.∀y.P(x,y), s_{0}, WeakQuantifier(∀y.P(s_{0},y), List((Atom(P(
    s_{0},s_{2})),s_{2})))) :- SkolemQuantifier(∀y.∃x.P(x,y), s_{2}, WeakQuantifier(∃
    x.P(x,s_{2}), List((Atom(P(s_{0},s_{2})),s_{0})))))
```

The LeanCoP interface only supports the `getExpansionSequent` method with exactly one formula in the succedent:

```
gapt> LeanCoP getExpansionSequent sequent map {toDeep(_)}
res24: Option[at.logic.gapt.proofs.HOLSequent] = Some(p(0), ((p(0)⊃p(s(0)))∧(p(s(0))
    ⊃p(s(s(0))))) :- p(s(s(0))))
```

You can also export sequents in TPTP format if you want to pass them to other provers manually:

```
gapt> TPTPFOLExporter.tptp_proof_problem_split(sequent)
res25: String =
"fof( formula0, axiom, 'p'('0') ).
fof( formula1, axiom, (! [X0] : ( 'p'(X0) => 'p'('s'(X0)) )) ).
fof( formula2, conjecture, 'p'('s'('s'('0'))) ).
"
```

4.5 Built-in tableaux prover

GAPT contains a built-in tableaux prover for propositional logic which can be called with the command `solve.solvePropositional`, for example as in:

```
gapt> solve.solvePropositional(parseFormula("a -> (b -> a&b)"))
res26: Option[at.logic.gapt.proofs.lkNew.LKProof] =
Some([p5] :- (a⊃(b⊃(a∧b))) (ImpRightRule(p4, Ant(0), Suc(0)))
[p4] a :- (b⊃(a∧b)) (ImpRightRule(p3, Ant(1), Suc(0)))
[p3] a, b :- (a∧b)  (AndRightRule(p1, Suc(0), p2, Suc(0)))
[p2] b :- b   (LogicalAxiom(b))
[p1] a :- a   (LogicalAxiom(a))
)
```

4.6 Cut-elimination (Gentzen's method)

The GAPT-system contains an implementation of reductive cut-elimination à la Gentzen. It can be used as follows: first we load a proof p with cuts (as in Appendix C).

```
gapt> val p = XMLProofDatabaseParser( "examples/simple/fol1.xml.gz" ).proofs(0)._2
p: at.logic.gapt.proofs.lkNew.LKProof =
[p25] ∀x.∀y.(P(x,y)⊃Q(x,y)) :- ∃x.∃y.(¬Q(x,y)⊃¬P(x,y)) (CutRule(p13, Suc(0), p24,
    Ant(0)))
[p24] ∀x.∃y.(¬P(x,y)∨Q(x,y)) :- ∃x.∃y.(¬Q(x,y)⊃¬P(x,y)) (ForallLeftRule(p23, Ant(0)
    , ∃y.(¬P(x,y)∨Q(x,y)), b, x))
[p23] ∃y.(¬P(b,y)∨Q(b,y)) :- ∃x.∃y.(¬Q(x,y)⊃¬P(x,y)) (ExistsLeftRule(p22, Ant(0), v
    , y))
[p22] (¬P(b,v)∨Q(b,v)) :- ∃x.∃y.(¬Q(x,y)⊃¬P(x,y)) (ExistsRightRule(p21, Suc(0), ∃y
    .(¬Q(x,y)⊃¬P(x,y)), b, x))
[p21] (¬P(b,v)∨Q(b,v)) :- ∃y.(¬Q(b,y)⊃¬P(b,y)) (ExistsRightRule(p20, Suc(0), (¬Q(b,
    y)⊃¬P(b,y)), v, y))
[p20] (¬P(b,v)∨Q(b,v)) :- (¬Q(b,v)⊃¬P(b,v)) (ImpRightRule(p19, Ant(1), Suc(0)))
```

```
[p19] (¬P(b,v)∨Q(b,v)), ¬Q(b,v) :- ¬P(b,v) (OrLeftRule(p16, Ant(0), p18, Ant(1)))
[p18] ¬Q(b,v), Q(b,v) :-  (NegLeftRule(p17, Suc(0)))
[p17] Q(b,v) ...
```

and then call the cut-elimination procedure:

```
gapt> val q = ReductiveCutElimination( p )
q: at.logic.gapt.proofs.lkNew.LKProof =
[p16] ∀x.∀y.(P(x,y)⊃Q(x,y)) :- ∃x.∃y.(¬Q(x,y)⊃¬P(x,y)) (ForallLeftRule(p15, Ant(0),
     ∀y.(P(x,y)⊃Q(x,y)), b, x))
[p15] ∀y.(P(b,y)⊃Q(b,y)) :- ∃x.∃y.(¬Q(x,y)⊃¬P(x,y)) (ForallLeftRule(p14, Ant(0), (P(
    b,y)⊃Q(b,y)), a, y))
[p14] (P(b,a)⊃Q(b,a)) :- ∃x.∃y.(¬Q(x,y)⊃¬P(x,y)) (ContractionRightRule(p13, Suc(0),
    Suc(1)))
[p13] (P(b,a)⊃Q(b,a)) :- ∃x.∃y.(¬Q(x,y)⊃¬P(x,y)), ∃x.∃y.(¬Q(x,y)⊃¬P(x,y)) (
    ImpLeftRule(p6, Suc(0), p12, Ant(0)))
[p12] Q(b,a) :- ∃x.∃y.(¬Q(x,y)⊃¬P(x,y)) (ExistsRightRule(p11, Suc(0), ∃y.(¬Q(x,y)⊃¬
    P(x,y)), b, x))
[p11] Q(b,a) :- ∃y.(¬Q(b,y)⊃¬P(b,y)) (ExistsRightRule(p10, Suc(0), (¬Q(b,y)⊃¬P(b,y)
    ), a, y))
[p10] Q(b,a) :- (¬Q(b,a)⊃¬P(b,a)) (ImpRightRule(p9, Ant(0), Suc(0)))
[p9] ¬Q(b,a), Q(b,a) :- ¬P(b,a) (WeakeningRightRul...
```

4.7 Skolemization

Skolemization consists of replacing the variables bound by strong quantifiers in the end-sequent of a proof by new function symbols thus obtaining a validity-equivalent sequent. In the GAPT-system Skolemization is implemented for proofs and can be used, e.g. as follows:

```
gapt> val Seq(x, y) = Seq("x", "y") map {FOLVar(_)}
x: at.logic.gapt.expr.FOLVar = x
y: at.logic.gapt.expr.FOLVar = y

gapt> val Pxy = FOLAtom("P", x, y)
Pxy: at.logic.gapt.expr.FOLAtom = P(x,y)

gapt> var p: LKProof = LogicalAxiom(Pxy)
p: at.logic.gapt.proofs.lkNew.LKProof =
[p1] P(x,y) :- P(x,y)  (LogicalAxiom(P(x,y)))

gapt> p = ExistsRightRule(p, Ex(x, Pxy), x)
p: at.logic.gapt.proofs.lkNew.LKProof = [p2] P(x,y) :- ∃x.P(x,y) (ExistsRightRule(p1,
     Suc(0), P(x,y), x, x))
[p1] P(x,y) :- P(x,y)  (LogicalAxiom(P(x,y)))

gapt> p = ForallLeftRule(p, All(y, Pxy), y)
p: at.logic.gapt.proofs.lkNew.LKProof = [p3] ∀y.P(x,y) :- ∃x.P(x,y) (ForallLeftRule(
    p2, Ant(0), P(x,y), y, y))
[p2] P(x,y) :- ∃x.P(x,y) (ExistsRightRule(p1, Suc(0), P(x,y), x, x))
```

```
[p1] P(x,y) :- P(x,y)  (LogicalAxiom(P(x,y)))

gapt> p = ForallRightRule(p, All(y, Ex(x, Pxy)), y)
p: at.logic.gapt.proofs.lkNew.LKProof = [p4] ∀y.P(x,y) :- ∀y.∃x.P(x,y) (
    ForallRightRule(p3, Suc(0), y, y))
[p3] ∀y.P(x,y) :- ∃x.P(x,y) (ForallLeftRule(p2, Ant(0), P(x,y), y, y))
[p2] P(x,y) :- ∃x.P(x,y) (ExistsRightRule(p1, Suc(0), P(x,y), x, x))
[p1] P(x,y) :- P(x,y)  (LogicalAxiom(P(x,y)))

gapt> p = ExistsLeftRule(p, Ex(x, All(y, Pxy)), x)
p: at.logic.gapt.proofs.lkNew.LKProof = [p5] ∃x.∀y.P(x,y) :- ∀y.∃x.P(x,y) (
    ExistsLeftRule(p4, Ant(0), x, x))
[p4] ∀y.P(x,y) :- ∀y.∃x.P(x,y) (ForallRightRule(p3, Suc(0), y, y))
[p3] ∀y.P(x,y) :- ∃x.P(x,y) (ForallLeftRule(p2, Ant(0), P(x,y), y, y))
[p2] P(x,y) :- ∃x.P(x,y) (ExistsRightRule(p1, Suc(0), P(x,y), x, x))
[p1] P(x,y) :- P(x,y)  (LogicalAxiom(P(x,y)))

gapt> val q = skolemize(p)
q: at.logic.gapt.proofs.lkNew.LKProof =
[p3] ∀y.P(s_{0},y) :- ∃x.P(x,s_{1}) (ForallLeftRule(p2, Ant(0), P(s_{0},y), s_{1}, y)
    )
[p2] P(s_{0},s_{1}) :- ∃x.P(x,s_{1}) (ExistsRightRule(p1, Suc(0), P(x,s_{1}), s_{0},
    x))
[p1] P(s_{0},s_{1}) :- P(s_{0},s_{1}) (LogicalAxiom(P(s_{0},s_{1})))
```

4.8 Interpolation

The command `ExtractInterpolant` extracts an interpolant from a sequent calculus proof which may contain atomic cuts and/or equality rules. Currently, we allow only reflexivity axioms, and axioms of the form $A \vdash A$; $\bot \vdash$, or $\vdash \top$. The implementation is based on Lemma 6.5 of [10]. The method expects a proof p and an arbitrary partition of the end-sequent $\Gamma \vdash \Delta$ of p into a "negative part" $\Gamma_1 \vdash \Delta_1$ and a "positive part" $\Gamma_2 \vdash \Delta_2$. It returns a formula I s.t. $\Gamma_1 \vdash \Delta_1, I$ and $I, \Gamma_2 \vdash \Delta_2$ are provable and I contains only such predicate symbols that appear in both, $\Gamma_1 \vdash \Delta_1$ and $\Gamma_2 \vdash \Delta_2$. For instance, suppose pr is the following proof:

$$\dfrac{\dfrac{P(a) \vdash P(a)}{a = b, P(a) \vdash P(a)}\,\text{(w:l)}}{a = b, P(a) \vdash P(b)}\,\text{=:r}$$

First, we construct the proof pr:

```
gapt> val ca = FOLConst( "a" )
ca: at.logic.gapt.expr.FOLConst = a

gapt> val cb = FOLConst( "b" )
cb: at.logic.gapt.expr.FOLConst = b
```

```
gapt> val pa = FOLAtom( "P", List( ca ) )
pa: at.logic.gapt.expr.FOLAtom = P(a)

gapt> val pb = FOLAtom( "P", List( cb ) )
pb: at.logic.gapt.expr.FOLAtom = P(b)

gapt> val aeqb = Eq( ca, cb )
aeqb: at.logic.gapt.expr.FOLAtom = a=b

gapt> val axpa = LogicalAxiom( pa )
axpa: at.logic.gapt.proofs.lkNew.LogicalAxiom =
[p1] P(a) :- P(a)  (LogicalAxiom(P(a)))

gapt> val axpb = LogicalAxiom( pb )
axpb: at.logic.gapt.proofs.lkNew.LogicalAxiom =
[p1] P(b) :- P(b)  (LogicalAxiom(P(b)))

gapt> val proof = WeakeningLeftRule( axpa, aeqb )
proof: at.logic.gapt.proofs.lkNew.WeakeningLeftRule =
[p2] a=b, P(a) :- P(a)  (WeakeningLeftRule(p1, a=b))
[p1] P(a) :- P(a)  (LogicalAxiom(P(a)))

gapt> val pr = EqualityRightRule( proof, aeqb, Suc( 0 ), pb )
pr: at.logic.gapt.proofs.lkNew.EqualityRightRule =
[p3] a=b, P(a) :- P(b)  (EqualityRightRule(p2, Ant(0), Suc(0), [2]))
[p2] a=b, P(a) :- P(a)  (WeakeningLeftRule(p1, a=b))
[p1] P(a) :- P(a)  (LogicalAxiom(P(a)))
```

In order to apply interpolation, we need to specify a partition of the end-sequent into $\Gamma_1 \vdash \Delta_1$ and $\Gamma_2 \vdash \Delta_2$, i.e. into the negative (npart) and positive (ppart) part, respectively. In this case, we set $\Delta_1 = \{P(b)\}$, $\Gamma_2 = \{a = b, P(a)\}$ and $\Gamma_1 = \Delta_2 = \emptyset$.

Then we can call `ExtractInterpolant( pr, npart, ppart )`, which returns the interpolant $I = (a = b \supset \neg P(a))$ of pr:

```
gapt> val I = ExtractInterpolant( pr, Seq( Suc( 0 ) ), Seq( Ant( 0 ), Ant( 1 ) ) )
I: at.logic.gapt.expr.HOLFormula = (a=b⊃¬P(a))
```

4.9 Expansion trees

Expansion trees are a compact representation of cut-free proofs. They have originally been introduced in [8]. GAPT contains an implementation of expansion trees for higher-order logic including functions for extracting expansion trees from proofs, for merging expansion trees, for pruning and transforming them in various ways and for viewing them in a comfortable way in the graphical user interface.

An expansion tree contains the instances of the quantifiers for a formula. In order to represent an LK-proof we use *expansion sequents*, i.e. sequents of expansion trees. We can obtain an expansion sequent for example by:

```
gapt> val p = SquareEdgesExampleProof( 4 )
p: at.logic.gapt.proofs.lkNew.LKProof =
[p40] ∀x.∀y.(P(x,y)⊃P(s(x),y)), P(0,0), ∀x.∀y.(P(x,y)⊃P(x,s(y))) :- P(s(s(s(s(0)))),
    s(s(s(s(0))))) (ContractionLeftRule(p39, Ant(0), Ant(2)))
[p39] ∀x.∀y.(P(x,y)⊃P(s(x),y)), P(0,0), ∀x.∀y.(P(x,y)⊃P(s(x),y)), ∀x.∀y.(P(x,y)⊃P(x
    ,s(y))) :- P(s(s(s(s(0)))),s(s(s(s(0))))) (ForallLeftRule(p38, Ant(0), ∀y.(P(x,y)
    ⊃P(s(x),y)), 0, x))
[p38] ∀y.(P(0,y)⊃P(s(0),y)), P(0,0), ∀x.∀y.(P(x,y)⊃P(s(x),y)), ∀x.∀y.(P(x,y)⊃P(x,s(
    y))) :- P(s(s(s(s(0)))),s(s(s(s(0))))) (ForallLeftRule(p37, Ant(0), (P(0,y)⊃P(s
    (0),y)), 0, y))
[p37] (P(0,0)⊃P(s(0),0)), P(0,0), ∀x.∀y.(P(x,y)⊃P(s(x),y)), ∀x.∀y.(P(x,y)⊃P(x,s(y)))
     :- P(s(s(s(s(0)))),s(s(s(s(0))))) (ImpLeftRule(p1, Suc(0), p36, Ant(1)))
[p36] ∀x.∀y.(P(x,y)⊃P(s(x),y)), P(s(0),0), ∀x.∀y.(P(x,y)⊃P(x,s(y))) :- P(s(s(s(s(0)
    ...
gapt> val E = LKToExpansionProof( p )
E: at.logic.gapt.proofs.expansionTrees.ExpansionSequent = WeakQuantifier(∀x.∀y.(P(x,y
    )⊃P(s(x),y)), List((WeakQuantifier(∀y.(P(0,y)⊃P(s(0),y)), List((Atom(P(0,0))⊃
    Atom(P(s(0),0)),0))),0), (WeakQuantifier(∀y.(P(s(0),y)⊃P(s(s(0)),y)), List((Atom(
    P(s(0),0))⊃Atom(P(s(s(0)),0)),0))),s(0)), (WeakQuantifier(∀y.(P(s(s(0)),y)⊃P(s(s
    (s(0))),y)), List((Atom(P(s(s(0)),0))⊃Atom(P(s(s(s(0))),0)),0))),s(s(0))), (
    WeakQuantifier(∀y.(P(s(s(s(0))),y)⊃P(s(s(s(s(0)))),y)), List((Atom(P(s(s(s(0)))
    ,0))⊃Atom(P(s(s(s(s(0)))),0)),0))),s(s(s(0)))))), Atom(P(0,0)), WeakQuantifier(∀x
    .∀y.(P(x,y)⊃P(x,s(y))), List((WeakQuantifier(∀y.(P(s(s(s(s(0)))),y)⊃P(s(s(s(s(0)
    ))),s(y))), List((Atom(P(s(s(s(s(0)))),0))⊃Atom(P(s(s(s(s(0)))),s(0))),0), (Atom(
    P(s(s(s(s(0)))),s(0)))⊃Atom(P(s(s(s(s(0)))),s(s(0)))),s(0)), (Atom(P(s(s(s(...
```

This expansion sequent can then be viewed in the graphical user interface by simply calling:

```
gapt> prooftool( E )
```

A window then opens that displays the end-sequent of `p`, i.e. the shallow sequent of `E`. You can then selectively expand quantifiers by clicking on them, see [6] for a detailed description.

4.10 Cut-elimination by resolution (CERES)

FiXme!

FiXme: Short example for first and higher-order CERES.

4.11 Cut-introduction

The cut-introduction algorithm as described in [5, 4, 3] is implemented in GAPT for introducing Π_1-cuts into a sequent calculus proof. We will use as input one of the proofs generated by the system, namely, `LinearExampleProof(9)`. But the user can also write his own proofs (see Section 3.2) and input them to the cut-introduction algorithm.

Take an example proof:

```
gapt> val p = LinearExampleProof(4)
p: at.logic.gapt.proofs.lkNew.LKProof =
```

```
[p18] ∀x.(P(x)⊃P(s(x))), P(0) :- P(s(s(s(s(0))))) (ContractionLeftRule(p17, Ant(0),
    Ant(2)))
[p17] ∀x.(P(x)⊃P(s(x))), P(0), ∀x.(P(x)⊃P(s(x))) :- P(s(s(s(s(0))))) (ForallLeftRule
    (p16, Ant(0), (P(x)⊃P(s(x))), 0, x))
[p16] (P(0)⊃P(s(0))), P(0), ∀x.(P(x)⊃P(s(x))) :- P(s(s(s(s(0))))) (ImpLeftRule(p1,
    Suc(0), p15, Ant(1)))
[p15] ∀x.(P(x)⊃P(s(x))), P(s(0)) :- P(s(s(s(s(0))))) (ContractionLeftRule(p14, Ant(0)
    , Ant(2)))
[p14] ∀x.(P(x)⊃P(s(x))), P(s(0)), ∀x.(P(x)⊃P(s(x))) :- P(s(s(s(s(0))))) (
    ForallLeftRule(p13, Ant(0), (P(x)⊃P(s(x))), s(0), x))
[p13] (P(s(0))⊃P(s(s(0)))), P(s(0)), ∀x.(P(x)⊃P(s(x))) :- P(s(s(s(s(0))))) (
    ImpLeftRule(p2, Suc(0), p12, Ant(1)))
[p12] ∀x.(P(x)⊃P(s(x))), P(s(s(0))) :- P(s(s(s(s(0))))) (ContractionLeftRule(p11...
```

Then compute a proof with a single cut that contains a single quantifier by:

```
gapt> val q = CutIntroduction.compressLKProof( p,
    DeltaTableMethod( manyQuantifiers=false ), verbose=true )
Total inferences in the input proof: 16
Quantifier inferences in the input proof: 4
End sequent: ∀x.(P(x)⊃P(s(x))), P(0) :- P(s(s(s(s(0)))))
Size of term set: 6
Smallest grammar of size 6:
Axiom: (τ)
Non-terminal vectors:
  (τ)
  (α_0)
Productions:
  τ -> -{P(0)}_a1

  τ -> -{∀x.(P(x)⊃P(s(x)))}_a0(s(α_0))

  τ -> -{∀x.(P(x)⊃P(s(x)))}_a0(α_0)

  τ -> {P(s(s(s(s(0)))))}_s0

  α_0 -> 0

  α_0 -> s(s(0))


Size of the canonical solution: 8
Size of the minimized solution: 4
CNF of minimized cut-formula number 0:
  P(α_0) :- P(s(s(α_0)))
Number of cuts introduced: 1
Total inferences in the proof with cut(s): 19
Quantifier inferences in the proof with cut(s): 5
q: Option[at.logic.gapt.proofs.lkNew.LKProof] =
Some([p19] ∀x.(P(x)⊃P(s(x))), P(0) :- P(s(s(s(s(0))))) (CutRule(p10, Suc(0), p18, Ant
```

```
    (0)))
[p18] ∀α_0.(P(α_0)⊃P(s(s(α_0)))), P(0) :- P(s(s(s(s(0))))) (ContractionLeftRule(p17,
     Ant(1), Ant(0)))
[p17] ∀α_0.(P(α_0)⊃P(s(s(α_0)))), ∀α_0.(P(α_0)⊃P(s(s(α_0)))), P(0) :- P(s(s(s(s(0))
    ))) (ForallLeftRule(p16, Ant(2), (P(α_0)⊃P(s(s(α_0)))), s(s(0)), α_0))
[p16] ∀α_0.(P(α_0)⊃P(s(s(α_0)))), P(0), (P(s(s(0)))⊃P(s(s(s(s(0)))))) :- P(s(s(s(s
    (0))))) (ForallLeftRule(p15, Ant(0), (P(α_0)⊃P(s(s(α_0)))), 0, α_0))
[p15] (P(0)⊃P(s(s(0)))), P(0), (P(s(s(0)))⊃P(s(s(s(s(0)))))) :- P(s(s(s(s(0))))) (
    ImpLeftRule(p11, Suc(0), p14, Ant(1)))
[p14] (P(s(s(0)))⊃P(s(s(s(s(0)))))), P(s(s(0))) :- P(s(s(s(s(0))))) (ImpLeftRule(p12,
     Suc(0), p13, Ant(0)))
[p13...
```

You can also try `DeltaTableMethod(manyQuantifiers=true)`, this will proceed as above but will compute a single cut with a block of quantifiers. The method `MaxSATMethod(1,2)` uses a reduction to a MaxSAT problem and an external MaxSAT-solver for finding a minimal grammar corresponding to a proof with a cut with two cuts, one with 1 quantifier, one with 2 quantifiers.

4.12 Tree grammars

The cut-introduction method described in Section 4.11 is based on the use of certain tree grammars for representing Herbrand-disjunctions. These are totally rigid acyclic tree grammars (TRATGs) and vectorial TRATGs (VTRATGs). As shown in [4], these grammars are intimately related to the structure of proofs with cuts. GAPT contains an implementation of these tree grammars, and given a finite tree language (i.e., a set of terms), is able to automatically find a (V)TRATG that covers this language:

```
gapt> val lang = 1 to 18 map { Numeral(_) }
lang: scala.collection.immutable.IndexedSeq[at.logic.gapt.expr.FOLTerm] = Vector(s(0)
    , s(s(0)), s(s(s(0))), s(s(s(s(0)))), s(s(s(s(s(0))))), s(s(s(s(s(0)))))), s(s(
    s(s(s(s(s(0))))))), s(s(s(s(s(s(s(0)))))))), s(s(s(s(s(s(s(s(0))))))))), s(s(
    s(s(s(s(s(s(s(s(s(0)))))))))), s(s(s(s(s(s(s(s(s(s(0))))))))))), s(s(s(s(s(s(
    s(s(s(s(s(0)))))))))))), s(s(s(s(s(s(s(s(s(s(s(s(0))))))))))))), s(s(s(s(s(s(
    s(s(s(s(s(s(s(0)))))))))))))), s(s(s(s(s(s(s(s(s(s(s(s(s(s(0))))))))))))))), s(s(
    s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(0)))))))))))))))), s(s(s(s(s(s(s(s(s(s(s(s(s(s(s
    (0)))))))))))))), s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(0))))))))))))))))))

gapt> val grammar = findMinimalGrammar(lang, 2)
grammar: at.logic.gapt.grammars.TratGrammar =
Axiom: τ
Non-terminals: τ, α_1, α_2
Productions:
  τ -> s(s(s(s(α_1))))
  τ -> s(α_1)
  α_1 -> s(s(α_2))
  α_1 -> s(α_2)
  α_1 -> α_2
  α_2 -> 0
```

```
  α_2 -> s(s(s(s(s(0))))))
  α_2 -> s(s(s(s(s(s(s(s(s(s(0)))))))))))

gapt> lang.toSet subsetOf grammar.language
res28: Boolean = true
```

You can also find minimal sub-grammars that still generate certain terms:

```
gapt> minimizeGrammar(grammar, Set(1,2,4,5) map {Numeral(_)})
res29: at.logic.gapt.grammars.TratGrammar =
Axiom: τ
Non-terminals: τ, α_1, α_2
Productions:
  τ -> s(s(s(s(α_1))))
  τ -> s(α_1)
  α_1 -> s(α_2)
  α_1 -> α_2
  α_2 -> 0
```

A Proof systems

A.1 LK

The rules of LK are listed below. Proof trees are constructed top-down, starting with axioms and with each rule introducing new inferences. With the exception of the definition rules, the constructors of the rules only allow inferences that are actually valid. Note that the rules are presented here as if they always act upon the outermost formulas in the upper sequent, but this is only for convenience of presentation. The basic constructors actually require the user to specify on which concrete formulas the inference should be performed.

Apart from those basic constructors, there is also a multitude of convenience constructors that facilitate easier proof construction. Moreover, there are so-called macro rules that reduce several inferences to a single command (e.g. introducing quantifier blocks). See the API documentation of the individual rules for details.

Axioms

$$\frac{}{A \vdash A}\ \text{(Logical axiom)} \qquad \frac{}{\vdash t = t}\ \text{(Reflexivity axiom)}$$

$$\frac{}{\vdash \top}\ \top\ \text{axiom} \qquad \frac{}{\bot \vdash}\ \bot\ \text{axiom}$$

$$\frac{}{\Gamma \vdash \Delta}\ \text{Theory axiom}$$

Logical axioms and theory axioms may only contain atomic formulas.

Cut

$$\frac{\Gamma \vdash \Delta, A \qquad A, \Sigma \vdash \Pi}{\Gamma, \Sigma \vdash \Delta, \Pi}\ \text{(cut)}$$

Structural rules

Left rules

$$\frac{\Gamma \vdash \Delta}{A, \Gamma \vdash \Delta}\ \text{(w:l)}$$

$$\frac{A, A, \Gamma \vdash \Delta}{A, \Gamma \vdash \Delta}\ \text{(c:l)}$$

Right rules

$$\frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta, A}\ \text{(w:r)}$$

$$\frac{\Gamma \vdash \Delta, A, A}{\Gamma \vdash \Delta, A}\ \text{(c:r)}$$

Propositional rules

Left rules

$$\frac{A, B, \Gamma \vdash \Delta}{A \wedge B, \Gamma \vdash \Delta}\ (\wedge\text{:l})$$

$$\frac{A, \Gamma \vdash \Delta \qquad B, \Sigma \vdash \Pi}{A \vee B, \Gamma, \Sigma \vdash \Delta, \Pi}\ (\vee\text{:l})$$

$$\frac{\Gamma \vdash \Delta, A}{\neg A, \Gamma \vdash \Delta}\ (\neg\text{:l})$$

$$\frac{\Gamma \vdash \Delta, A \qquad B, \Sigma \vdash \Pi}{A \supset B, \Gamma, \Sigma \vdash \Delta, \Pi}\ (\supset\text{:l})$$

Right rules

$$\frac{\Gamma \vdash \Delta, A \qquad \Sigma \vdash \Pi, B}{\Gamma, \Sigma \vdash \Delta, \Pi, A \wedge B}\ (\wedge\text{:r})$$

$$\frac{\Gamma \vdash \Delta, A, B}{\Gamma \vdash \Delta, A \vee B}\ (\vee\text{:r})$$

$$\frac{A, \Gamma \vdash \Delta}{\Gamma \vdash \Delta, \neg A}\ (\neg\text{:r})$$

$$\frac{A, \Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, A \supset B}\ (\supset\text{:r})$$

Quantifier rules

Left rules

$$\frac{A[t/x], \Gamma \vdash \Delta}{\forall x A, \Gamma \vdash \Delta}\ (\forall\text{:l})$$

$$\frac{A[y/x], \Gamma \vdash \Delta}{\exists x A, \Gamma \vdash \Delta}\ (\exists\text{:l})$$

Right rules

$$\frac{\Gamma \vdash \Delta, A[y/x]}{\Gamma \vdash \Delta, \forall x A}\ (\forall\text{:r})$$

$$\frac{\Gamma \vdash \Delta, A[t/x]}{\Gamma \vdash \Delta, \exists x A}\ (\exists\text{:r})$$

The variable y must not occur free in Γ, Δ or A. The term t must avoid variable capture, i.e. it must not contain free occurrences of variables bound in A.

Equality rules

Left rules

$$\frac{s = t, A[T/s], \Sigma \vdash \Pi}{s = t, A[T/t], \Sigma \vdash \Pi}\ (=\text{:l})$$

$$\frac{s = t, A[T/t], \Sigma \vdash \Pi}{s = t, A[T/s], \Sigma \vdash \Pi}\ (=\text{:l})$$

Right rules

$$\frac{s = t, \Sigma \vdash \Pi, A[T/s]}{s = t, \Sigma \vdash \Pi, A[T/t]}\ (=\text{:r})$$

$$\frac{s = t, \Sigma \vdash \Pi, A[T/t]}{s = t, \Sigma \vdash \Pi, A[T/s]}\ (=\text{:r})$$

Each equation rule replaces exactly one occurrence of a term in its auxiliary formula.

Definition rules

$$\frac{A, \Gamma \vdash \Delta}{B, \Gamma \vdash \Delta}\,(\text{def:l})\quad \frac{\Gamma \vdash \Delta, A}{\Gamma \vdash \Delta, B}\,(\text{def:r})$$

These definition rules are extremely liberal, as they allow the replacement of any formula by any other formula.

Induction

The induction rule applies to arbitrary algebraic data types. Let $c_1, \ldots, c_n$ be the constructors of a type and let k_i be the arity of c_i. Let $F[x]$ be a formula with x a free variable of the appropriate type. Then we call the sequent $\mathcal{S}_i := F[x_1], \ldots, F[x_{k_i}], \Gamma_i \vdash \Delta_i, F[c_i(x_1, \ldots, x_{k_i})]$ the i-th induction step. In this case, the induction rule has the form

$$\frac{\begin{array}{cccc}(\pi_1) & (\pi_2) & & (\pi_n) \\ \mathcal{S}_1 & \mathcal{S}_2 & \ldots & \mathcal{S}_n\end{array}}{\Gamma \vdash \Delta, \forall x F[x]}\,(\text{ind})$$

In the case of the natural numbers, there are two constructors: 0 of arity 0 and s of arity 1. Consequently, the induction rule reduces to

$$\frac{\begin{array}{cc}(\pi_1) & (\pi_2) \\ \Gamma_1 \vdash \Delta_1, F[0] & F[x], \Gamma_2 \vdash \Delta_2, F[sx]\end{array}}{\Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2, \forall x F[x]}\,(\text{ind})$$

A.2 Resolution

Initial clauses

$$\frac{}{C}\,\text{InputClause}$$

$$\frac{}{t = t}\,\text{ReflexivityClause}$$

$$\frac{}{a \vee \neg a}\,\text{TautologyClause}$$

Structural rules

$$\frac{a \vee a \vee C}{a \vee C}\,\text{Factor}$$

$$\frac{C}{C\sigma}\,\text{Instance}$$

Logical rules

$$\frac{C \vee a \qquad \neg a \vee D}{C \vee D}\ \text{Resolution}$$

$$\frac{C \vee t = s \qquad l[t] \vee D}{C \vee l[s] \vee D}\ \text{Paramodulation}$$

$$\frac{C \vee t = s \qquad l[s] \vee D}{C \vee l[t] \vee D}\ \text{Paramodulation}$$

A.3 LKsk

LKsk is a sequent calculus used in the CERES method for higher-order logic, see [11] for an overview of both the calculus and the method.

LKsk operates on labelled formulas. A label ℓ is a finite list of terms, a labelled formula is a pair (F, ℓ), written as $\langle F \rangle^{\ell}$. If ℓ is a label and t a term, we write $\ell; t$ for the concatenation of ℓ and t.

Axioms

$$\frac{}{\langle A \rangle^{\ell_1} \vdash \langle A \rangle^{\ell_2}}\ \text{(Logical axiom)} \qquad \frac{}{\vdash \langle t = t \rangle^{\ell}}\ \text{(Reflexivity axiom)}$$

$$\frac{}{\vdash \langle \top \rangle^{\ell}}\ \top\ \text{axiom} \qquad \frac{}{\langle \bot \rangle^{\ell} \vdash}\ \bot\ \text{axiom}$$

$$\frac{}{\langle A_1 \rangle^{\ell_1}, \ldots, \langle A_m \rangle^{\ell_m} \vdash \langle B_1 \rangle^{\ell'_1}, \ldots, \langle B_n \rangle^{\ell'_n}}\ \text{Theory axiom}$$

Logical axioms and theory axioms may only contain atomic formulas.

Cut

$$\frac{\Gamma \vdash \Delta, \langle A \rangle^{\ell_1} \qquad \langle A \rangle^{\ell_2}, \Sigma \vdash \Pi}{\Gamma, \Sigma \vdash \Delta, \Pi}\ \text{(cut)}$$

Note that the labels of the cut formulas do not need to be equal.

Structural rules

Left rules

$$\frac{\Gamma \vdash \Delta}{\langle A \rangle^{\ell}, \Gamma \vdash \Delta}\ \text{(w:l)} \qquad \frac{\langle A \rangle^{\ell}, \langle A \rangle^{\ell}, \Gamma \vdash \Delta}{\langle A \rangle^{\ell}, \Gamma \vdash \Delta}\ \text{(c:l)}$$

Right rules

$$\frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta, \langle A \rangle^\ell}\,(\mathsf{w{:}r})$$

$$\frac{\Gamma \vdash \Delta, \langle A \rangle^\ell, \langle A \rangle^\ell}{\Gamma \vdash \Delta, \langle A \rangle^\ell}\,(\mathsf{c{:}r})$$

Propositional rules

Left rules

$$\frac{\langle A \rangle^\ell, \langle B \rangle^\ell, \Gamma \vdash \Delta}{\langle A \wedge B \rangle^\ell, \Gamma \vdash \Delta}\,(\wedge\mathsf{{:}l})$$

$$\frac{\langle A \rangle^\ell, \Gamma \vdash \Delta \qquad \langle B \rangle^\ell, \Sigma \vdash \Pi}{\langle A \vee B \rangle^\ell, \Gamma, \Sigma \vdash \Delta, \Pi}\,(\vee\mathsf{{:}l})$$

$$\frac{\Gamma \vdash \Delta, \langle A \rangle^\ell}{\langle \neg A \rangle^\ell, \Gamma \vdash \Delta}\,(\neg\mathsf{{:}l})$$

$$\frac{\Gamma \vdash \Delta, \langle A \rangle^\ell \qquad \langle B \rangle^\ell, \Sigma \vdash \Pi}{\langle A \supset B \rangle^\ell, \Gamma, \Sigma \vdash \Delta, \Pi}\,(\supset\mathsf{:l})$$

Right rules

$$\frac{\Gamma \vdash \Delta, \langle A \rangle^\ell \qquad \Sigma \vdash \Pi, \langle B \rangle^\ell}{\Gamma, \Sigma \vdash \Delta, \Pi, \langle A \wedge B \rangle^\ell}\,(\wedge\mathsf{{:}r})$$

$$\frac{\Gamma \vdash \Delta, \langle A \rangle^\ell, \langle B \rangle^\ell}{\Gamma \vdash \Delta, \langle A \vee B \rangle^\ell}\,(\vee\mathsf{{:}r})$$

$$\frac{\langle A \rangle^\ell, \Gamma \vdash \Delta}{\Gamma \vdash \Delta, \langle \neg A \rangle^\ell}\,(\neg\mathsf{{:}r})$$

$$\frac{\langle A \rangle^\ell, \Gamma \vdash \Delta, \langle B \rangle^\ell}{\Gamma \vdash \Delta, \langle A \supset B \rangle^\ell}\,(\supset\mathsf{:r})$$

Quantifier rules

Left rules

$$\frac{\langle A[t/x] \rangle^{\ell;t}, \Gamma \vdash \Delta}{\langle \forall x A \rangle^\ell, \Gamma \vdash \Delta}\,(\forall\mathsf{{:}l})$$

$$\frac{\langle A[y/x] \rangle^\ell, \Gamma \vdash \Delta}{\langle \exists x A \rangle^\ell, \Gamma \vdash \Delta}\,(\exists\mathsf{{:}l})$$

$$\frac{\langle A[f(\ell)/x] \rangle^\ell, \Gamma \vdash \Delta}{\langle \exists x A \rangle^\ell, \Gamma \vdash \Delta}\,(\exists^{sk}\mathsf{{:}l})$$

Right rules

$$\frac{\Gamma \vdash \Delta, \langle A[y/x] \rangle^\ell}{\Gamma \vdash \Delta, \langle \forall x A \rangle^\ell}\,(\forall\mathsf{{:}r})$$

$$\frac{\Gamma \vdash \Delta, \langle A[f(\ell)/x] \rangle^\ell}{\Gamma \vdash \Delta, \langle \forall x A \rangle^\ell}\,(\forall^{sk}\mathsf{{:}r})$$

$$\frac{\Gamma \vdash \Delta, \langle A[t/x] \rangle^{\ell;t}}{\Gamma \vdash \Delta, \langle \exists x A \rangle^\ell}\,(\exists\mathsf{{:}r})$$

The variable y must not occur free in Γ, Δ or A. The term t must avoid variable capture, i.e. it must not contain free occurrences of variables bound in A.

Equality rules

Left rules **Right rules**

$$\frac{\langle s = t\rangle^{\ell_1}, \langle A[T/s]\rangle^{\ell_2}, \Sigma \vdash \Pi}{\langle s = t\rangle^{\ell_1}, \langle A[T/t]\rangle^{\ell_2}, \Sigma \vdash \Pi}\ (=\text{:l}) \qquad \frac{\langle s = t\rangle^{\ell_1}, \Sigma \vdash \Pi, \langle A[T/s]\rangle^{\ell_2}}{\langle s = t\rangle^{\ell_1}, \Sigma \vdash \Pi, \langle A[T/t]\rangle^{\ell_2}}\ (=\text{:r})$$

$$\frac{\langle s = t\rangle^{\ell_1}, \langle A[T/t]\rangle^{\ell_2}, \Sigma \vdash \Pi}{\langle s = t\rangle^{\ell_1}, \langle A[T/s]\rangle^{\ell_2}, \Sigma \vdash \Pi}\ (=\text{:l}) \qquad \frac{\langle s = t\rangle^{\ell_1}, \Sigma \vdash \Pi, \langle A[T/t]\rangle^{\ell_2}}{\langle s = t\rangle^{\ell_1}, \Sigma \vdash \Pi, \langle A[T/s]\rangle^{\ell_2}}\ (=\text{:r})$$

Each equation rule replaces exactly one occurrence of a term in its auxiliary formula.

A.4 Ral

Ral is a higher-order resolution calculus used in the CERES method for higher-order logic, see again [11] for details.

Ral operates on sequents of labelled formulas, just as LKsk. A label ℓ is a finite set of terms, a labelled formula is a pair (F, ℓ), written as $\langle F\rangle^{\ell}$. If ℓ is a label and t a term, we write $\ell; t$ for the concatenation of ℓ and t.

$$\frac{}{\Gamma \vdash \Delta}\ \text{RalInitial}$$

The substitution also substitutes the variables in the labels:

$$\frac{\Gamma \vdash \Delta}{\Gamma\sigma \vdash \Delta\sigma}\ \text{RalSub}$$

The resolution rule does not require the labels to be equal:

$$\frac{\Gamma \vdash \Delta, \langle A\rangle^{\ell_1}, \ldots, \langle A\rangle^{\ell_m} \qquad \langle A\rangle^{\ell_1}, \ldots, \langle A\rangle^{\ell_n}, \Sigma \vdash \Pi}{\Gamma, \Sigma \vdash \Delta, \Pi}\ \text{RalCut}$$

Factoring is restricted to atoms:

$$\frac{\langle a\rangle^{\ell_1} a, \langle a\rangle^{\ell_2}, \Gamma \vdash \Delta}{\langle a\rangle^{\ell_1}, \Gamma \vdash \Delta}\ \text{RalFactor} \qquad \frac{\Gamma \vdash \Delta, \langle a\rangle^{\ell_1}, \langle a\rangle^{\ell_2}}{\Gamma \vdash \Delta, \langle a\rangle^{\ell_1}}\ \text{RalFactor}$$

Paramodulation can rewrite the equation in both directions, in a formula of the antecedent or the succedent—this is one of the four possibilities:

$$\frac{\Gamma \vdash \Delta, \langle t = s\rangle^{\ell} \qquad \Sigma \vdash \Pi, \langle A[t]\rangle^{\ell}}{\Gamma, \Sigma \vdash \Delta, \langle A[s]\rangle^{\ell}}\ \text{RalPara}$$

$$\frac{\langle \forall x\ A[x]\rangle^{\ell}, \Gamma \vdash \Delta}{\langle A[s\ell]\rangle^{\ell}, \Gamma \vdash \Delta}\ \text{RalAllF} \qquad \frac{\Gamma \vdash \Delta, \langle \forall x\ A[x]\rangle^{\ell}}{\Gamma \vdash \Delta, \langle A[\alpha]\rangle^{\ell;\alpha}}\ \text{RalAllT}$$

$$\frac{\langle \exists x\ A[x]\rangle^{\ell}, \Gamma \vdash \Delta}{\langle A[\alpha]\rangle^{\ell;\alpha}, \Gamma \vdash \Delta}\ \text{RalExF} \qquad \frac{\Gamma \vdash \Delta, \langle \exists x\ A[x]\rangle^{\ell}}{\Gamma \vdash \Delta, \langle A[s\ell]\rangle^{\ell}}\ \text{RalExT}$$

$$\frac{\langle \top\rangle^{\ell}, \Gamma \vdash \Delta}{\Gamma \vdash \Delta}\ \text{RalTopF} \qquad \frac{\Gamma \vdash \Delta, \langle \bot\rangle^{\ell}}{\Gamma \vdash \Delta}\ \text{RalBottomT}$$

$$\frac{\langle \neg A\rangle^{\ell}, \Gamma \vdash \Delta}{\Gamma \vdash \Delta, \langle A\rangle^{\ell}}\ \text{RalNegF} \qquad \frac{\Gamma \vdash \Delta, \langle \neg A\rangle^{\ell}}{\langle A\rangle^{\ell}, \Gamma \vdash \Delta}\ \text{RalNegT}$$

$$\frac{\langle A \wedge B\rangle^{\ell}, \Gamma \vdash \Delta}{\langle A\rangle^{\ell}, \langle B\rangle^{\ell}, \Gamma \vdash \Delta}\ \text{RalAndF} \qquad \frac{\Gamma \vdash \Delta, \langle A \wedge B\rangle^{\ell}}{\Gamma \vdash \Delta, \langle A\rangle^{\ell}}\ \text{RalAndT1} \qquad \frac{\Gamma \vdash \Delta, \langle A \wedge B\rangle^{\ell}}{\Gamma \vdash \Delta, \langle B\rangle^{\ell}}\ \text{RalAndT2}$$

$$\frac{\langle A \vee B\rangle^{\ell}, \Gamma \vdash \Delta}{\langle A\rangle^{\ell}, \Gamma \vdash \Delta}\ \text{RalOrF1} \qquad \frac{\langle A \vee B\rangle^{\ell}, \Gamma \vdash \Delta}{\langle B\rangle^{\ell}, \Gamma \vdash \Delta}\ \text{RalOrF2} \qquad \frac{\Gamma \vdash \Delta, \langle A \vee B\rangle^{\ell}}{\Gamma \vdash \Delta, \langle A\rangle^{\ell}, \langle B\rangle^{\ell}}\ \text{RalOrT}$$

$$\frac{\langle A \supset B\rangle^{\ell}, \Gamma \vdash \Delta}{\Gamma \vdash \Delta, \langle A\rangle^{\ell}}\ \text{RalImpF1} \qquad \frac{\langle A \supset B\rangle^{\ell}, \Gamma \vdash \Delta}{\langle B\rangle^{\ell}, \Gamma \vdash \Delta}\ \text{RalImpF2} \qquad \frac{\Gamma \vdash \Delta, \langle A \supset B\rangle^{\ell}}{\langle A\rangle^{\ell}, \Gamma \vdash \Delta, \langle B\rangle^{\ell}}\ \text{RalImpT}$$

FiXme!

FiXme: Expansion trees?

B Prover9 term parser (parseFormula)

The Prover9 parser interprets names strings built over the alphabet $[\mathtt{a}, \ldots, \mathtt{z}, \mathtt{A}, \ldots, \mathtt{Z}, \mathtt{0}, \ldots \mathtt{9}]$. Also the equality symbol $=$ and the arithmetic operators $\{+, -, *, /\}$ are names.

Since prover 9 has two naming schemes, we describe both of them. The default is the native LADR scheme.

B.1 LADR naming scheme (default)

Names starting with lower case letters from $\mathtt{u}$ to $\mathtt{z}$ are interpreted as variable symbols. Every name starting with a different letter is interpreted as a constant/function/predicate symbol, depending on the position in the term.

B.2 Prolog naming scheme

Names starting with upper case letters from $\mathtt{A}$ to $\mathtt{Z}$ are interpreted as variable symbols. Every name starting with a different letter is interpreted as a constant/function, depending on the position in the term. Predicates start with a lower or an upper case letter.

B.3 Terms and Formulas

Arguments are always separated by commas and put in parenthesis. Terms are built from variable, constant symbols and function symbols applied to a number of arguments, which are terms. Likewise, atoms are predicate symbols, possibly with arguments which are again terms. The function symbols $\{+, -, *, /\}$ and the equality predicate $=$ are written in infix notation.

For example, the following are terms in the LADR naming scheme:

```
"constant", "variable", "function(constant,variable)", "s_{21} + 5"
```

And the following are atoms:

```
"P1(alpha, beta(x))", "25 = v"
```

The logical connectives are called $\{\&, |, ->, \texttt{all}, \texttt{exists}\}$ where the quantifiers and negation are written in prefix form and conjunction, disjunction and implication are written in infix form.

Conjunction and disjunction are right-associative. Quantifiers always need to be enclosed in parenthesis but a series of quantifiers doesn't need parenthesis in between. The binding order is (from strongest to weakest): $\neg, \forall, \exists, \wedge \vee, \rightarrow$.

For example, the formula $\forall x(P(x, f(x)) \rightarrow \exists y P(x, y))$ is parsed as:

```
gapt> parseFormula("(all x (P(x,f(x)) -> (exists y P(x,y))))")
res30: at.logic.gapt.expr.FOLFormula = ∀x.(P(x,f(x))⊃∃y.P(x,y))
```

The Prolog scheme can be parsed as well using the `Prover9TermParser` class:

```
gapt> Prover9TermParser.parseFormula("(all X (P(X,f(X)) -> (exists Y P(X,Y))))")
res31: at.logic.gapt.expr.FOLFormula = ∀X.(P(X,f(X))⊃∃Y.P(X,Y))
```

C XML proof parser

This method `XMLProofDatabaseParser` will take as an argument a string that represents the path of a file containing a proof database in the XML format (generated by HLK, for example) and will return a proof database containing a list of pairs. It expects a file (the string of a proof will not work) and you can use the relative path. In the list returned, each pair is composed of a string and an object representing a proof within the system. The string is the name of the proof defined in the XML file. For example:

```
gapt> val proofs = XMLProofDatabaseParser( "examples/simple/fol1.xml.gz" ).proofs
proofs: List[(String, at.logic.gapt.proofs.lkNew.LKProof)] =
List((p,[p25] ∀x.∀y.(P(x,y)⊃Q(x,y)) :- ∃x.∃y.(¬Q(x,y)⊃¬P(x,y)) (CutRule(p13, Suc(0)
    , p24, Ant(0)))
[p24] ∀x.∃y.(¬P(x,y)∨Q(x,y)) :- ∃x.∃y.(¬Q(x,y)⊃¬P(x,y)) (ForallLeftRule(p23, Ant(0)
    , ∃y.(¬P(x,y)∨Q(x,y)), b, x))
[p23] ∃y.(¬P(b,y)∨Q(b,y)) :- ∃x.∃y.(¬Q(x,y)⊃¬P(x,y)) (ExistsLeftRule(p22, Ant(0), v
    , y))
[p22] (¬P(b,v)∨Q(b,v)) :- ∃x.∃y.(¬Q(x,y)⊃¬P(x,y)) (ExistsRightRule(p21, Suc(0), ∃y
    .(¬Q(x,y)⊃¬P(x,y)), b, x))
```

```
[p21] (¬P(b,v)∨Q(b,v)) :- ∃y.(¬Q(b,y)⊃¬P(b,y)) (ExistsRightRule(p20, Suc(0), (¬Q(b,
    y)⊃¬P(b,y)), v, y))
[p20] (¬P(b,v)∨Q(b,v)) :- (¬Q(b,v)⊃¬P(b,v)) (ImpRightRule(p19, Ant(1), Suc(0)))
[p19] (¬P(b,v)∨Q(b,v)), ¬Q(b,v) :- ¬P(b,v) (OrLeftRule(p16, Ant(0), p18, Ant(1)))
[p18] ¬Q(b,v), Q(b,v) :-  (NegLeftRul...
```

returns a list of length 1 as shown by entering

```
gapt> proofs.length
res32: Int = 1
```

Its only element is `proofs(0)`, the name of this proof can be obtained by entering

```
gapt> proofs(0)._1
res33: String = p
```

and the proof itself by

```
gapt> val proof = proofs(0)._2
proof: at.logic.gapt.proofs.lkNew.LKProof =
[p25] ∀x.∀y.(P(x,y)⊃Q(x,y)) :- ∃x.∃y.(¬Q(x,y)⊃¬P(x,y)) (CutRule(p13, Suc(0), p24,
    Ant(0)))
[p24] ∀x.∃y.(¬P(x,y)∨Q(x,y)) :- ∃x.∃y.(¬Q(x,y)⊃¬P(x,y)) (ForallLeftRule(p23, Ant(0)
    , ∃y.(¬P(x,y)∨Q(x,y)), b, x))
[p23] ∃y.(¬P(b,y)∨Q(b,y)) :- ∃x.∃y.(¬Q(x,y)⊃¬P(x,y)) (ExistsLeftRule(p22, Ant(0), v
    , y))
[p22] (¬P(b,v)∨Q(b,v)) :- ∃x.∃y.(¬Q(x,y)⊃¬P(x,y)) (ExistsRightRule(p21, Suc(0), ∃y
    .(¬Q(x,y)⊃¬P(x,y)), b, x))
[p21] (¬P(b,v)∨Q(b,v)) :- ∃y.(¬Q(b,y)⊃¬P(b,y)) (ExistsRightRule(p20, Suc(0), (¬Q(b,
    y)⊃¬P(b,y)), v, y))
[p20] (¬P(b,v)∨Q(b,v)) :- (¬Q(b,v)⊃¬P(b,v)) (ImpRightRule(p19, Ant(1), Suc(0)))
[p19] (¬P(b,v)∨Q(b,v)), ¬Q(b,v) :- ¬P(b,v) (OrLeftRule(p16, Ant(0), p18, Ant(1)))
[p18] ¬Q(b,v), Q(b,v) :-  (NegLeftRule(p17, Suc(0)))
[p17] Q(b...
```

You can then view this proof in the graphical user interface `prooftool` by entering

```
gapt> prooftool( proof )
```

In the folder `../examples/simple` you can find a number of further simple proofs.

References

[1] Josep Argelich, Chu Min Li, Felip Manya, and Jordi Planes. The first and second max-SAT evaluations. *Journal on Satisfiability, Boolean Modeling and Computation*, 4(2-4):251–278, 2008.

[2] Matthias Baaz and Alexander Leitsch. Cut-elimination and Redundancy-elimination by Resolution. *Journal of Symbolic Computation*, 29(2):149–176, 2000.

[3] Stefan Hetzl, Alexander Leitsch, Giselle Reis, Janos Tapolczai, and Daniel Weller. Introducing Quantified Cuts in Logic with Equality. In Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *Automated Reasoning - 7th International Joint Conference, IJCAR*, volume 8562 of *Lecture Notes in Computer Science*, pages 240–254. Springer, 2014.

[4] Stefan Hetzl, Alexander Leitsch, Giselle Reis, and Daniel Weller. Algorithmic introduction of quantified cuts. *Theoretical Computer Science*, 549:1–16, 2014.

[5] Stefan Hetzl, Alexander Leitsch, and Daniel Weller. Towards algorithmic cut-introduction. In *Proceedings of the 18th international conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'12, pages 228–242, Berlin, Heidelberg, 2012. Springer-Verlag.

[6] Stefan Hetzl, Tomer Libal, Martin Riener, and Mikheil Rukhaia. Understanding Resolution Proofs through Herbrand's Theorem. In Didier Galmiche and Dominique Larchey-Wendling, editors, *Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX) 2013, Proceedings*, volume 8123 of *Lecture Notes in Computer Science*, pages 157–171. Springer, 2013.

[7] William McCune. Prover9 and mace4 manual - input files, 2005–2010. https://www.cs.unm.edu/~mccune/mace4/manual/2009-11A/input.html.

[8] Dale Miller. A Compact Representation of Proofs. *Studia Logica*, 46(4):347–370, 1987.

[9] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima Inc, 2011.

[10] Gaisi Takeuti. *Proof Theory*. North-Holland, Amsterdam, 2nd edition, March 1987.

[11] Daniel Weller. *CERES in higher-order logic*. 2010. Wien, Techn. Univ., Diss., 2010.